Introduction to Rcpp

Irina Gaynanova
Guest lecturer: Yunfeng Zhang

3/28/2019

Some useful Rcpp references

- **Official vignettes** for Rcpp package
- **H.Wickham's Advanced R**
- **H.Wickham's R packages**
- **Codes and slides**:
  https://github.com/Pennisetum/Rcpp-class

Why C++?

- **Rcpp** package greatly simplifies interface from R to C++ (no simplifying interface for Fortran or C to my knowledge)
- fast and elegant linear algebra using companion packages, i.e. **RcppArmadillo**
- easy to incorporate within the R package with Rstudio

```
install.packages("Rcpp")
```

Note: most likely, you have everything already installed. If not, please come to next office hours for trouble-shooting.

Compilers

For Mac users:

- g++ is no longer available since 2011
- Xcode has to be installed (may also require **gfortran**)

For Windows users:

- Things get tricky
- Use **Rtools** package

Simplest C++

Let's write a function that determines whether the given number is even or odd

```
isOddR <- function(num){
  result <- num %% 2 == 1
  return(result)
}
isOddR(10)
```

```
## [1] FALSE
```

```
isOddR(13)
```

```
## [1] TRUE
```

Simplest C++

Let's now try to write this function in C++ via Rcpp

```
library(Rcpp)
cppFunction("
bool isOddCpp(int num){
  bool result = (num%2 == 1);
  return result;
}")
isOddCpp(10)
```

```
## [1] FALSE
```

```
isOddCpp(13)
```

```
## [1] TRUE
```

Simplest C++

```
library(Rcpp)
cppFunction("
bool isOddCpp(int num){
  bool result = (num%2 == 1);
  return result;
}")
```

- **cppFunction()** compiles, links and then imports corresponding code into R
- **isOddCpp()** is written in C++:
  1. Both the type of return value (**bool**) and the type of each argument (**int**, **bool**) have to be specified
  2. Each command is separated by **;**
  3. Some commands may have slighly different syntax (%% versus %)

Basic variable types of Rcpp

cpp	Rcpp
int	int
float	float
double	double
bool	bool
char	char

Moderate C++

What if we have tons of C++ code? Should we wrap it all manually?

Moderate C++

What if we have tons of C++ code? Should we wrap it all manually?

No. Save as C++ file and then source from within R.

Moderate C++

What if we have tons of C++ code? Should we wrap it all manually?

No. Save as C++ file and then source from within R.

In Rstudio, File $->$ New File $->$ C++ file

Moderate C++

What if we have tons of C++ code? Should we wrap it all manually?

No. Save as C++ file and then source from within R.

In Rstudio, File $->$ New File $->$ C++ file

Rstudio is smart and tries to make your job easier. What do you see in the .cpp file?

.cpp files for Rcpp

```
#include <Rcpp.h>
using namespace Rcpp;
```

This includes Rcpp header and states we are using Rcpp namespace so we don't need to write Rcpp::NumericVector

```
// This is a simple example of exporting a C++ function to
```

This is a comment within C++, starts with //

```
// [[Rcpp::export]]
```

Comes before the function that we want to be usable within R when we source the corresponding cpp file

```
NumericVector x
```

NumericVector is an Rcpp type for a vector that has numeric arguments (as the name suggests)

.cpp files for Rcpp

Save the file as Test.cpp. Use

```
library(Rcpp)
sourceCpp("Test.cpp")
```

```
##
## > timesTwo(42)
## [1] 84
```

```
x <- c(1,2,3)
timesTwo(x)
```

```
## [1] 2 4 6
```

Note that two things happened: R code within cpp file run, and extra R code run

Variable types: vectors

- In R: **c( )**; in Rcpp:
  - **IntegerVector** vectors of integers
  - **NumericVector** vectors of numeric values
  - **LogicalVector** vectors of logical values
- No additional code required if return vectors
- For cpp users: **NumericVector** can be converted into **std::vector**

Variable types: vectors

- Dangerous zone!!!
  - Copy a vector by clone
  - Reason: they are pointers

Variable types: vectors

```
library(Rcpp)
sourceCpp("copy.cpp")
x <- seq(1.0, 3.0, by=1)
x
```

```
## [1] 1 2 3
```

```
cbind(x, cpy(x))
```

```
##                x
## [1,] 0.0000000 0.0000000
## [2,] 0.6931472 0.6931472
## [3,] 1.0986123 1.0986123
```

Variable types: vectors

```
library(Rcpp)
sourceCpp("copy.cpp")
x <- seq(1.0, 3.0, by=1)
x
```

```
## [1] 1 2 3
```

```
cbind(x, cpy2(x))
```

```
##      x
## [1,] 1 0.0000000
## [2,] 2 0.6931472
## [3,] 3 1.0986123
```

Other variable types

R	Rcpp
matrix	NumericMatrix
strings	CharacterVector
list	List

Rcpp sugar

- Do we need write all the basic functions?
  - No! Use Rcpp API: sugar
  - Write cpp code like R

```
library(Rcpp)
sourceCpp("sugar.cpp")
x <- 1:5
y <- rep(4, 5)
plus(x, y)
```

```
## [1] 5 6 7 8 9
```

```
seq_len(5)
```

```
## [1] 1 2 3 4 5
```

Bootstrap example

We would like to calculate confidence interval around the sample mean using **bootstrap**

Want to take B samples **with replacement** from given data $x \in \mathbb{R}^n$, calculate mean/st.dev on each sample

Bootstrap example

```r
# ds - vector of observations
# B - number of bootstrap samples
bootstrap_r <- function(ds, B = 1000){
  boot_stat <- matrix(NA, nrow = B, ncol = 2)
  n <- length(ds)
  # Perform bootstrap
  for(i in 1:B) {
    # Create a sample of size n with replacement
    gen_data <- ds[sample(n, n, replace=TRUE)]
    # Calculate sample data mean and SD
    boot_stat[i,] <- c(mean(gen_data),sd(gen_data))
  }
  return(boot_stat)
}
```

Bootstrap example

Let's check how this works in practice

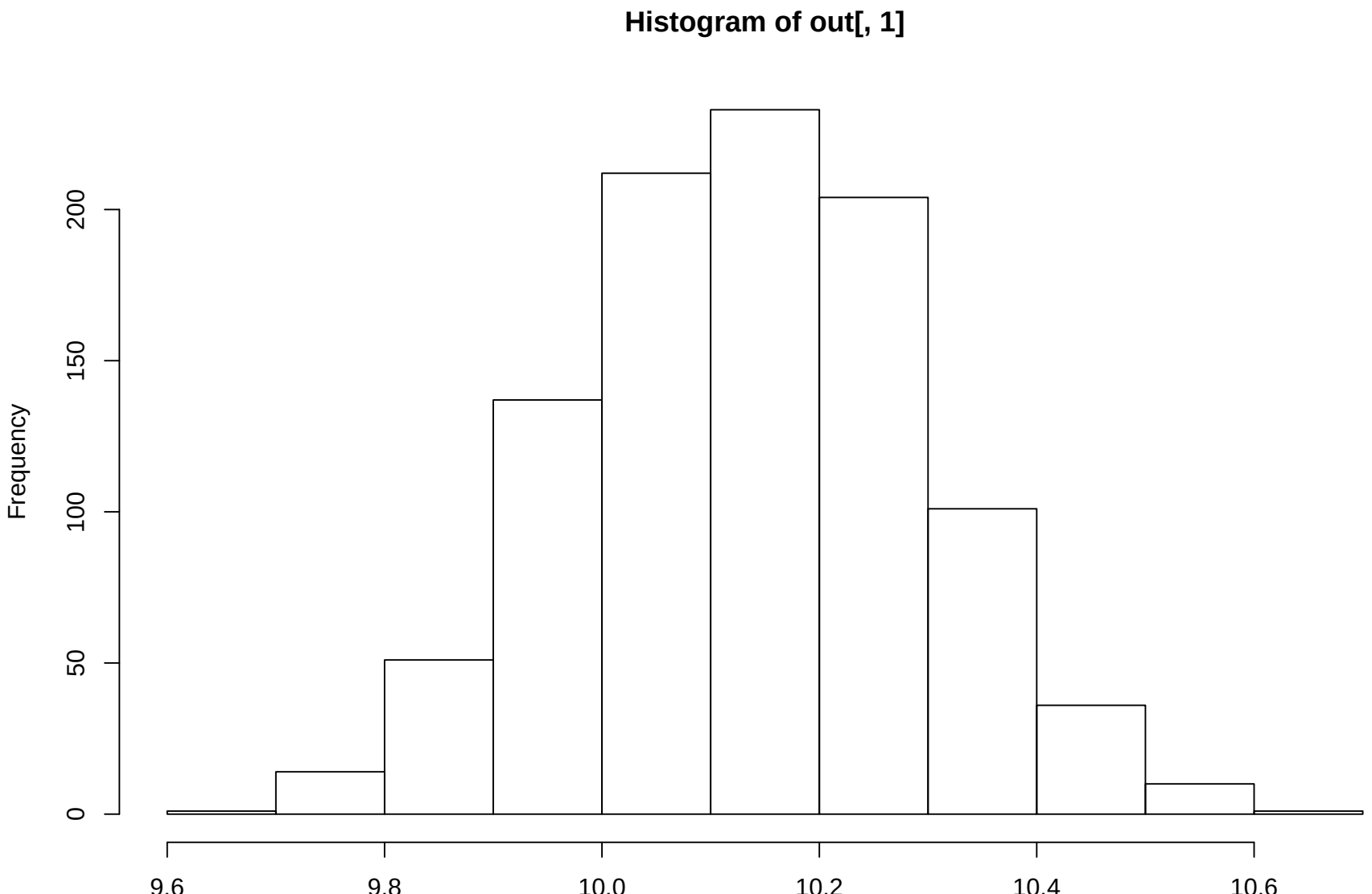
```
ds <- rnorm(1000, mean = 10, sd = 5)
out <- bootstrap_r(ds)
library(microbenchmark)
microbenchmark(bootstrap_r(ds), times = 10)
```

```
## Unit: milliseconds
##              expr      min       lq     mean   median
##  bootstrap_r(ds) 62.60064 64.85492 65.76513 66.07046 67.
##  neval
##     10
```

Bootstrap example

Distribution of mean over samples (truth 10)

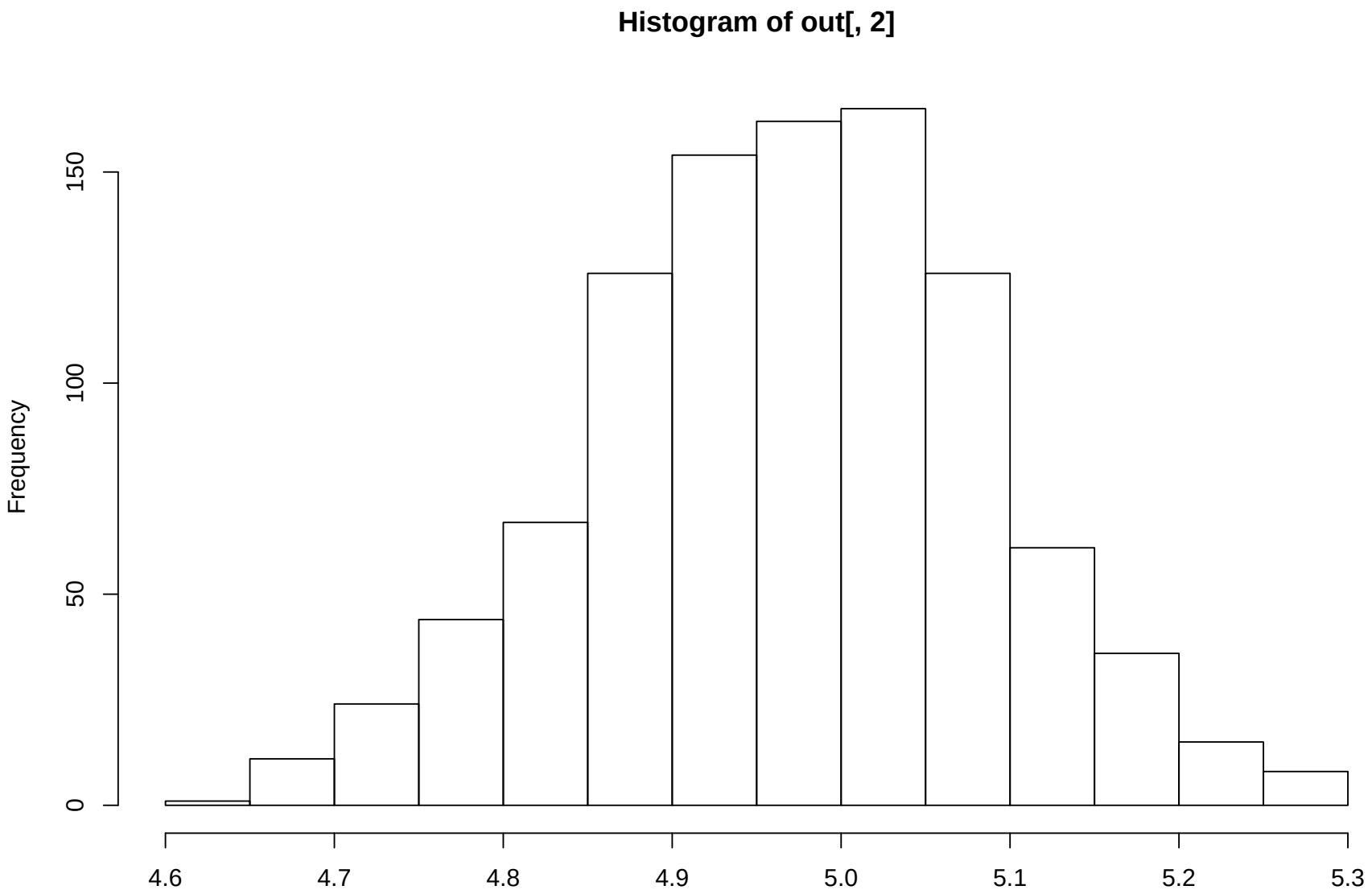
```
hist(out[, 1])
```



Bootstrap example

Distribution of sd over samples (truth 5)

```
hist(out[, 2])
```



Transfering the code to C++

```cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericMatrix bootstrap_cpp(NumericVector ds,
int B = 1000) {
// Preallocate storage for statistics
NumericMatrix boot_stat(B, 2); // Number of observations
  int n = ds.size();
  // Perform bootstrap
for(int i = 0; i < B; i++) {
 // Sample initial data
  NumericVector gen_data = ds[ floor(runif(n, 0, n)) ];
  // Calculate sample mean and std dev
  boot_stat(i, 0) = mean(gen_data);
  boot_stat(i, 1) = sd(gen_data);
}
// Return bootstrap results
return boot_stat;
```

Transfering the code to C++

- **NumericMatrix** - Rcpp matrix type
- Indexing goes from 0 to n-1 (rather than from 1 to n)
- ds**.size()** is used as length command
- i++ means i gets increase by 1 at each loop
- floor(runif(n,0,n)) - only return from 0 to n-1, consistent with indexinv
- matrix indexing using (,) rather than [,]

Bootstrap code in C++ vs R

```
library(Rcpp)
sourceCpp("Bootstrap.cpp")
set.seed(2308)
ds <- rnorm(1000, mean = 10, sd = 5)
set.seed(34)
outR <- bootstrap_r(ds)
set.seed(34)
outCpp <- bootstrap_cpp(ds)
all.equal(outR, outCpp)
```

```
## [1] TRUE
```

Dreaded for loop

```
library(microbenchmark)
microbenchmark(bootstrap_cpp(ds), bootstrap_r(ds), times =
```

```
## Unit: milliseconds
##                 expr      min       lq     mean   median
##  bootstrap_cpp(ds) 19.40186 21.40975 25.29927 22.65633 2
##    bootstrap_r(ds) 58.07435 63.57397 67.02008 66.09466 6
##  neval
##     50
##     50
```

Linear Algebra via RcppArmadillo

```
install.packages("RcppArmadillo")
```

When writing C++ code, this will require the use of different header, i.e.

```
#include <RcppArmadillo.h>
```

Armadillo clickable reference

Matrix multiplication with Rcpp Armadillo

```
#include <RcppArmadillo.h>
using namespace Rcpp;

// [[Rcpp::export]]
arma::mat matrix_mult(const arma::mat& X, const arma::mat&
  int m = X.n_rows;
  int n = Y.n_cols;
  arma::mat Z(m,n);
  Z = X * Y;
  return Z;
}
```

- **arma::mat** - matrix class within Armadillo library
- **& X** - uses pointer rather than copying the whole matrix
- **.n_rows()** and **.n_cols()** allow to get dimensions

Matrix multiplication

```
library(RcppArmadillo)
```

```
## Warning: package 'RcppArmadillo' was built under R versi
```

```
sourceCpp("ArmadilloExamples.cpp")

X = matrix(rnorm(300), 30, 10)
Y = matrix(rnorm(200), 10, 20)
prodCpp = matrix_mult(X, Y)
prodR = X%*%Y
all.equal(prodCpp, prodR)
```

```
## [1] TRUE
```

Linear model fit

- **ArmadilloExamples.cpp** contains another function that fits linear model
- **arma::colvec** - vector class within Armadillo, again use **&** to avoid copying
- **arma::solve(X, y)** - similar to R solve, here it calculates $(X^\top X)^{-1} X^\top Y$
- **std::innerproduct** - inner product function within std namespace (standard library in C++), 4 arguments: beginning and end of 1st vector, beginning of 2nd vector, initial value
- **.begin()** returns iterator pointing to the 1st element of the vector
- **.end()** returns iterator pointing to the last element of the vector
- **X.t()** is matrix transpose
- **pinv** - Moore-Penrose pseudo-inverse (generalized inverse based on SVD)
- **Rcpp::list** - list class within Rcpp package

Rcpp and RcppArmadillo - summary

- Do not try to memorize all C/C++ classes/commands - rather learn how to search for what you need and how to learn from examples you can find
- Some good references are in the beginning of the slides as well as within Armadillo library